Parallel Computing with OpenMP to solve matrix Multiplication

Rifat Chowdhury

UCONN BIOGRID REU Summer 2010

Department of Computer Science & Engineering

University of Connecticut, Storrs, CT 06269

Abstract: The purpose of this summer research is to design an algorithm through the means of C programming language using the libraries OpenMP to compute Matrix Multiplication efficiently. Using the libraries of OpenMP and some of its functions, we can optimize the speed up of the algorithm we are to write which will compute the matrix multiplication. We first start by writing a simple program which computes a predetermined matrix and shows the results. On the Bio-grid server, we then compile and execute our code. At this point it still uses a single core to compute the matrix multiplication. Later on, we add the openMP library and add some OpenMP functions to our code to parallelize the computation; which will allow more than one core of a processor, in our case, the Bio-grid head node computer to use all its four cores to compute the resulting matrix. We then test the run time of our program, we add a timer function to our code which keeps track of the time it took for the Bio-grid head node computer to do the parallelization. We do so First without the openMP code, and then with the OpenMP code. We will conduct separate trials for each input matrix size. We will systematically record each matrix input size and conduct 5 trials for each input size and record the time it took for the Bio-grid Head node to parallelize the matrix multiplication. The Bio-grid cluster has a head node computer and 11 child node computers. In order to access and utilize the cores of the child node computers to compute matrix multiplication we need to add the MPI library to our code. As mentioned earlier, in our experimental data we will have two separate categories. The first one is a simple matrix multiplication code. The second category is openMP with matrix multiplication. This will allow us to compare the speedup of our algorithm when more than one core of a processor is used.

Key Words: Multi-core Architecture, Parallelization, Multi-threading, openMP

2 Parallel Computing with OpenMP to solve matrix Multiplication

1. Introduction

1.1 Multi-core architectures

A processor is basically a unit that reads and executes program instructions, which are fixed-length typically 32 or 64 bit or variable-length chunks of data. The data in the instruction tells the processor what to do. The instructions are very basic things like reading data from memory or sending data to the user display, but they are processed so rapidly that we experience the results as the smooth operation of a program.

A core is the part of the processor which performs reading and executing of the instruction. Single core processors can only execute one instruction at a time. However as the name implies, Multi-core processors are composed of more than one core. A very common example would be a dual core processor. The advantage of a multi-core processor over a single core one is that the multi-core processor can either use both its cores to accomplish a single task or it can span threads which divided tasks between both its cores, so that it takes twice the amount of time it would take to execute the task than it would on a single core processor. Multi- core processors can also execute multiple tasks at a single time. A common example would be watching a movie on windows media player while your dual-core processor is running a background virus check. Multi-core is a shared memory processor. All cores share the same memory. All cores are on the same chip on a multi-core architecture.

1.2 OpenMP library

The OpenMP library is an API (Application Programming Interface) for writing shared memory parallel applications in programming languages such as C, C++ and FORTRAN. This API consists of compiler directives, runtime routines, and Environmental variables. Some the advantages of OpenMP includes: good performance, portable (it is supported by a large number of compilers), requires very little programming effort and allows the program to be parallelized incrementally. OpenMP is widely available and used, mature, lightweight, and ideally suited for multi-core architectures. Data can be shared or private in the OpenMP memory model. When data is private it is visible to one thread only, when data is public it is global and visible to all threads. OpenMP divides tasks into threads; a thread is the smallest unit of a processing that can be scheduled by an operating system. The master thread assigns tasks unto worker threads. Afterwards, they execute the task in parallel using the multiple cores of a processor.

3 Parallel Computing with OpenMP to solve matrix Multiplication

2. Our approach to solve the problem

We first familiarized ourselves with the concepts of parallel programming before jumping into the details. We first browsed through the OpenMP official website to learn about multi-core architecture, how parallel processes are run on a multi-core architecture. Then we took a sample C program which executes a sample matrix multiplication problem. We then took that matrix code and modified it by adding the OpenMP libraries and OpenMP functions. We further added a timer function to calculate the total time it took for the processor to compute the final matrix. We then ran into a problem, which was we were unable to give a matrix input beyond 1000*1000. We ran into an error called "Segmentation fault". So we had to dynamically allocate memory for the matrix and then free the space we created to solve this problem so that the program, when it is ran, accepts matrix sizes of very large inputs. This mechanism is necessary for our purposes because with large input matrix sizes it is clearer to see the differences in speedup of our OpenMP algorithm versus matrix multiplication without openMP algorithm.

3. Experimental Results and analysis

3.1 runtime comparison

Table 1: Total runtime to compute final matrix without openMP (part 1)

Trial #	(500*500)*(500*500)	(700*700)*(700*700)	(1000*1000)*(1000*1000)
1	1.52 seconds	4.24 seconds	12.64 seconds
2	1.51 seconds	4.28 seconds	12.65 seconds
3	1.59 seconds	4.30 seconds	12.64 seconds
4	1.56 seconds	4.27 seconds	12.57 seconds
5	1.55 seconds	4.28 seconds	12.62 seconds
Average runtime:	1.546 seconds	4.274 seconds	12.624 seconds
Worse case Runtime:	1.59 seconds	4.30 seconds	12.65 seconds

4 Parallel Computing with OpenMP to solve matrix Multiplication

Table 2: Total runtime to compute final matrix without openMP (part2)

Trial #	(1200*1200)*(1200*1200)	(2000*2000)*(2000*2000)	(3000*3000)*(3000*3000)
1	24.69 seconds	156.28 seconds	542.79 seconds
2	24.43 seconds	156.14 seconds	542.96 seconds
3	24.31 seconds	156.20 seconds	542.97 seconds
4	24.57 seconds	156.08 seconds	543.08 seconds
5	24.44 seconds	156.24 seconds	543.95 seconds
Average runtime:	24.488 seconds	156.188 seconds	543.15 seconds
Worst case Runtime:	24.69 seconds	156.28 seconds	543.95 seconds

Table 3: Total runtime to compute final matrix with openMP (part 1)

Trial #	(500*500)*(500*500)	(700*700)*(700*700)	(1000*1000)*(1000*1000)
1	0.53 seconds	1.38 seconds	3.79 seconds
2	0.55 seconds	1.81 seconds	3.80 seconds
3	0.51 seconds	1.35 seconds	3.80 seconds
4	0.55 seconds	1.38 seconds	3.78 seconds
5	0.54 seconds	1.35 seconds	3.84 seconds
Average Runtime:	0.536 seconds	1.454 seconds	3.802 seconds
Worse Case Runtime:	0.55 seconds	1.81 seconds	3.84 seconds

5 Parallel Computing with OpenMP to solve matrix Multiplication

Table 4: Total runtime to compute final matrix with openMP (part 2)

Trial #	(1200*1200)* (1200*1200)	(2000*2000)* (2000*2000)	(3000*3000)* (3000*3000)
1	7.38 seconds	44.00 seconds	152.02 seconds
2	7.70 seconds	44.04 seconds	152.02 seconds
3	7.45 seconds	44.01 seconds	152.00 seconds
4	7.49 seconds	44.00 seconds	151.96 seconds
5	7.38 seconds	44.01 seconds	152.04 seconds
Average runtime:	7.48 seconds	44.012 seconds	152.008 seconds
Worse case Runtime:	7.70 seconds	44.04 seconds	152.04 seconds

Table 5: Speedup calculations based on average runtime

Matrix input size(n*n)*(n*n)	Without OpenMP	OpenMP	Speedup increase
500*500	1.546 seconds	0.536 seconds	2.884x
700*700	4.274 seconds	1.454 seconds	2.938x
1000*1000	12.624 seconds	3.802 seconds	3.320x
1200*1200	24.488 seconds	7.48 seconds	3.274x
2000*2000	156.188 seconds	44.012 seconds	3.549x
3000*3000	543.15 seconds	152.008 seconds	3.573x

6 Parallel Computing with OpenMP to solve matrix Multiplication

Table 6: Speedup calculations based on Worse case runtime

Matrix input size(n*n)*(n*n)	Without OpenMP	OpenMP	Speedup increase
500*500	1.59 seconds	0.55 seconds	2.891x
700*700	4.30 seconds	1.81 seconds	2.376x
1000*1000	12.65 seconds	3.84 seconds	3.294x
1200*1200	24.69 seconds	7.70 seconds	3.206x
2000*2000	156.28 seconds	44.04 seconds	3.549x
3000*3000	543.95 seconds	152.04 seconds	3.578x

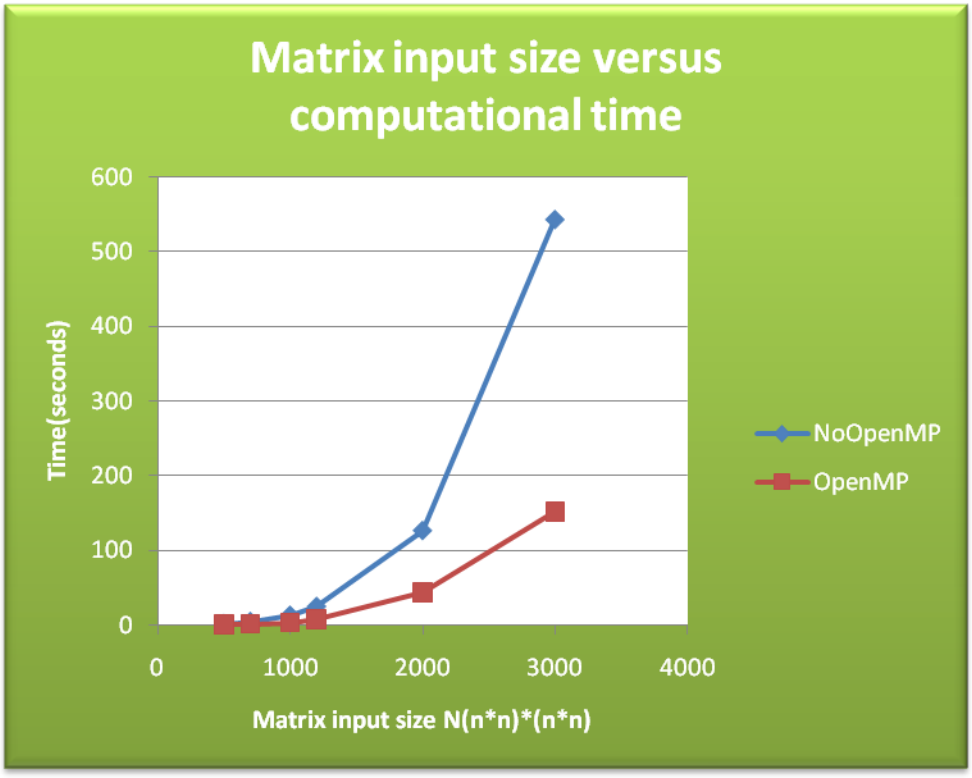

Figure 1: OpenMP versus without OpenMP final matrix computational time (Average case)

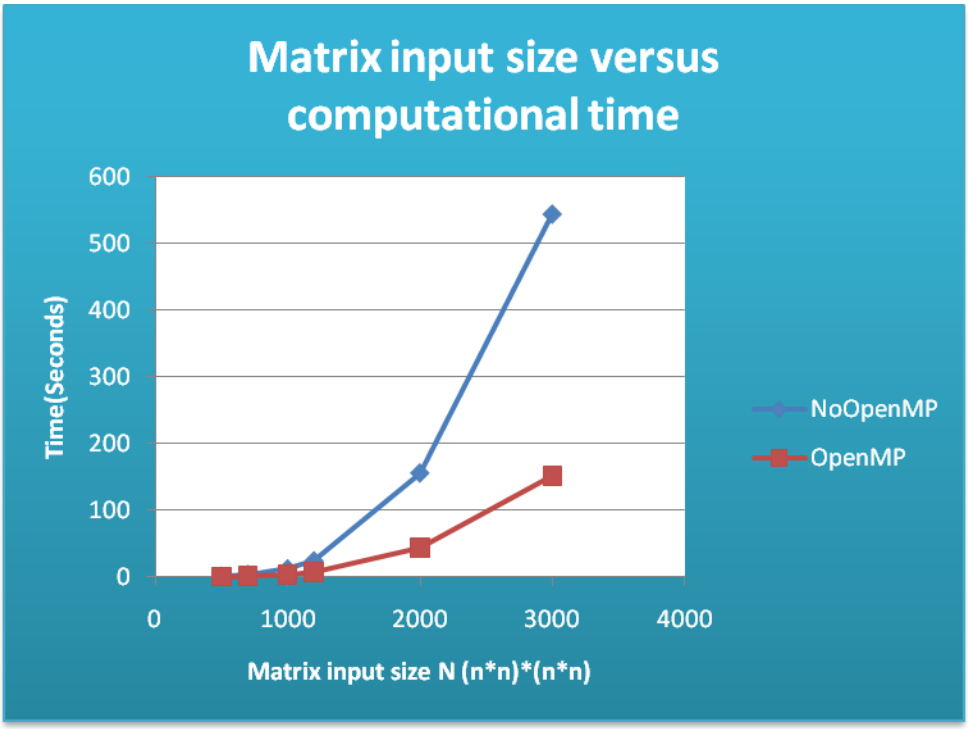

Figure2: OpenMP versus without OpenMP final matrix computational time (worse case)

3.4 further analysis

It should be well noted that with openMP, you can set the number of threads to any specified number you want. For example, you can set the number of threads to one. An interesting scenario arises; the computation of the result matrix takes longer using OpenMP with one thread than it does without OpenMP. Another interesting point to note, as we have seen based on the results, as we increase the number of threads, with OpenMP, the speedup is significantly faster than in comparison to using only four threads. When more than four threads are used, the result matrix is computed sequentially rather than in parallel. With four threads, each core is assigned a thread with OpenMP. However, with more than four threads, each core is assigned multiple threads.

4. Conclusion

Clearly, as the input size is increased, the difference between the total times it takes the Bio-grid head node computer to compute the final matrix with the OpenMP code and without the OpenMP code increases significantly. As expected the speedup is up to four times, with the

small amount of trials we have conducted our best speedup was about 3.578 times, which is almost close to the ideal 4 times using four cores of a processor to compute the final matrix instead of one. Had we chosen to include even a larger matrix, based on the graphs and tables, the speedup would have been as close to 4 times. Therefore, as the number of input size approaches infinity the speedup of our algorithm with OpenMP in comparison to without OpenMP would approach 4 times, as expected.

5. Future work

If we were to extend this project, we would add the MPI library to our existing code. We will add some MPI functions to send the matrix initialization tasks and computation tasks to multiple processors. For comparison purposes, first we would just get a working version without openMP library. Using the Bio-grid cluster, we can use up to 11 processors to compute the resulting matrix. In the best case, this should be up to 11 times faster than matrix multiplication without OpenMP code. We would then rename that C file and call it Matrix multiplication with MPI only.

Once we are successful, we can now add OpenMP to our MPI code and now this will allow each processor to utilize its four cores to compute the final result matrix. In the best case scenario, our speedup would be 44 times faster than matrix multiplication without OpenMP or MPI. Each of the processor in the Bio-grid Cluster contains four cores. This is how in the best case 11*4 =44, we may be able to observe a speedup up to Forty-four times faster than computing the result matrix without OpenMP or MPI.

Once this is established, we will test the run time of our program like previously. We then increase the input size of our Matrix. We will then compare results of speedup of our code in regards to single core, using Only OpenMP, MPI Only and using a hybrid parallelization of OpenMP and MPI.

Now we extend our problem from Matrix multiplication to a computational biology problem. Using our algorithm, we will identify “Boolean networks and related biological networks based on Matrix multiplication and fingerprint function.”[1] The Bio-Grid research advisor, Dr .Huang, will give me research papers to read and help me understand and analyze this problem. This is to be done under independent study which fulfills a professional requirement for an undergraduate student prior to graduating from the University of Connecticut in the Computer Science department!

6. Special thanks to NSF REU grant

I would like to personally acknowledge the opportunity National Science Foundation (NSF) has given Bio-grid students to acquire research experience which is very beneficial and helpful towards developing the academic and research skills for the student. With this research experience, the student has an advantage applying for Graduate Schools to further extend this research to contribute to the field of Computer Science and beyond. As research continues, the possibilities of newer discoveries are endless! Truly, NSF sponsored Research Experience for Undergraduates (REU) program will only encourage students to pursue higher education beyond the bachelor's degree. I am very thankful for the grant and the experience. I hope this research experience will end up benefitting myself first in the long run and also, benefit society in some way or another.

7. References

1. **JOURNAL OF COMPUTATIONAL BIOLOGY Volume 7, Numbers 3/4, 2000 Mary Ann Liebert, Inc. Pp. 331–343** Algorithms for Identifying Boolean Networks and Related Biological Networks Based on Matrix Multiplication and Fingerprint Function TATSUYA AKUTSU, SATORU MIYANO, and SATORU KUHARA

2. http://en.wikipedia.org/wiki/Thread_%28computer_science%29 December 2009

3. http://en.wikipedia.org/wiki/Parallel_computing

4. http://en.wikipedia.org/wiki/Multi-core_processor June 2009

5. "Basic Concepts in Parallelization" Ruud Van der Pas Pp. 1-58 http://www.compunity.org/training/tutorials/2%20Basic_Concepts_Parallelization.pdf

6. "An OverView of OpenMP" Ruud Van der Pas pp.1-161

   http://www.compunity.org/training/tutorials/3%20Overview_OpenMP.pdf

7. "An Overview of OpenMP 3.0" Ruud Van Der Pas pp1-107

   https://iwomp.zih.tu-dresden.de/downloads/2.Overview_OpenMP.pdf

8. Barbic, Jernej "Multi-core Architectures" May 3 2007

9. Nadgir, Neelakanth, June 2001 http://developers.sun.com/solaris/articles/openmp.html

10. Barney, Blaise 06/28/05

    https://computing.llnl.gov/tutorials/openMP/samples/C/omp_mm.c

8. Appendix/additional data collected during experiment

NoOpenMP + matrix multiplication

1000*1000 matrix takes 13 seconds to compute

2000*2000 matrix takes 156 seconds to compute

1200*1200 matrix takes 25 seconds to compute

700*700 matrix takes 5 seconds to compute

1300*1300 matrix takes 36 seconds to compute

1500*1500 matrix takes 63 seconds to compute

3000*3000 matrix takes 513 seconds to compute

OpenMP +matrix Multiplication(with printing message: (thread x did row y))(overall synchronization time)

1000*1000 matrix takes 4 seconds to compute

2000*2000 matrix takes 44 seconds to compute

1200*1200 matrix takes 8 seconds to compute

700*700 matrix takes 1 second to compute

1300*1300 matrix takes 10 seconds to compute

1500*1500 matrix takes 18 seconds to compute

OpenMP+matrix Multiplication(no printed message) (overall synchronization time)

1000*1000 matrix takes 4 seconds to compute

2000*2000 matrix takes 44 seconds to compute

1200*1200 matrix takes 7 seconds to compute

700*700 matrix takes 2 seconds to compute

1300*1300 matrix takes 11 seconds to compute

1500*1500 matrix takes 18 seconds to compute

11 Parallel Computing with OpenMP to solve matrix Multiplication

3000*3000 matrix takes 152 seconds to compute

1000*1000 takes 3.98 seconds using 200 threads!

3000*3000 takes 148.17 seconds using 200 threads!

source code (Matrix Multiplication without OpenMP)

```
/*objective:
matrix multiplication in dynamic way
*/
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <float.h>

int main()
{

int **a,**b,**c;
//int c[3][3];
int a_r,a_c,b_r,b_c;
clock_t start,end;            /* variables to store time difference between
                                        start of paralleization and end of
parallelization */
double dif;                   /*variable to calculate the time difference
between the parallelization */
int i,j,k;
//clrscr();
again:
printf("\nenter rows and columns for matrix one:");
scanf("%d%d",&a_r,&a_c);
printf("\nenter rows and columns for matrix two:");
scanf("%d%d",&b_r,&b_c);
if(a_c!=b_r )
{
printf("\ncan not multiply");
goto again;
}
/* allocate memory for matrix one */
a=(int **) malloc(10*a_r);
for( i=0;i<a_c; i++)
{
a[i]=(int *) malloc(10*a_c);
}

/* allocate memory for matrix two */
b=(int **) malloc(10*b_r);
for( i=0;i<b_c; i++)
```

```
{
b[i]=(int *) malloc(10*b_c);
}
/* allocate memory for sum matrix */
c=(int **) malloc(10*a_r);
for( i=0;i< b_c; i++)
{
c[i]=(int *) malloc(10*b_c);
}
    printf("Initializing matrices...\n");
      printf("Hello");
 start =clock();  //start the timer


//initializing first matrix
for(i=0;i<a_r; i++)
{
for(j=0;j<a_c; j++)
{
a[i][j] = i+j;
}
}

// initializing second matrix
for(i=0;i<b_r; i++)
{
for(j=0;j<b_c; j++)
{
b[i][j] = i*j;
}
}

/*initialize product matrix */
for(i=0;i<a_r; i++)
{
for(j=0;j< b_c; j++)
{
c[i][j]=0;
}
}

/* multiply matrix one and matrix two */
for(i=0;i<a_r; i++)
{
for(j=0;j<a_c; j++)
{
for(k=0;k<b_c; k++)
{
c[i][j]=c[i][j]+a[i][k]*b[k][j];
}
}
}
```

13 Parallel Computing with OpenMP to solve matrix Multiplication

```
printf("*******************************************************\n");
printf ("Done.\n");
  end= clock();   //end the timer
  dif = ((double) (end - start)) / CLOCKS_PER_SEC; //store the difference

 printf("Parallelization took %f sec. time.\n", dif);
/*free memory*/
for(i=0;i<a_r; i++)
{
free(a[i]);
}
free(a);
for(i=0;i<a_c; i++)
{
free(b[i]);
}
free(b);
for(i=0;i<b_c; i++)
{
free(c[i]);
}
free(c);


}
```

source code (Matrix multiplication + OpenMP)

```
/*objective:
matrix multiplication in dynamic way
*/
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{

int **a,**b,**c;        // variables to store allocated memory
int a_r,a_c,b_r,b_c, nthreads, tid, chunk =10;  //variables to input matrix
size and variables to be used by OpenMP functions
double dif;                    //variable to calculate the time difference
between the parallelization
int i,j,k;                     // variables to be used in for loops to
generate matrices

again:
printf("\nenter rows and columns for matrix one:");
scanf("%d%d",&a_r,&a_c);
printf("\nenter rows and columns for matrix two:");
scanf("%d%d",&b_r,&b_c);
if(a_c!=b_r )
{
```

```
printf("\ncan not multiply");
goto again;
}
/* allocate memory for matrix one */
a=(int **) malloc(10*a_r);
for( i=0;i<a_c; i++)
{
a[i]=(int *) malloc(10*a_c);
}

/* allocate memory for matrix two */
b=(int **) malloc(10*b_r);
for( i=0;i<b_c; i++)
{
b[i]=(int *) malloc(10*b_c);
}
/* allocate memory for sum matrix */
c=(int **) malloc(10*a_r);
for( i=0;i< b_c; i++)
{
c[i]=(int *) malloc(10*b_c);
}
    printf("Initializing matrices...\n");
      printf("Hello\n");
  double start = omp_get_wtime( );  //start the timer
/*** Spawn a parallel region explicitly scoping all variables ***/
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
 {
      tid = omp_get_thread_num();
  if (tid == 0)
    {
      nthreads = omp_get_num_threads();
    printf("Starting matrix multiple example with %d threads\n",nthreads);
    }

//initializing first matrix
#pragma omp for schedule (static, chunk)
for(i=0;i<a_r; i++)
{
for(j=0;j<a_c; j++)
{
a[i][j] = i+j;
}
}

// initializing second matrix
#pragma omp for schedule (static, chunk)
for(i=0;i<b_r; i++)
{
for(j=0;j<b_c; j++)
{
b[i][j] = i*j;
}
```

```
}

/*initialize product matrix */
#pragma omp for schedule (static, chunk)
for(i=0;i<a_r; i++)
{
for(j=0;j< b_c; j++)
{
c[i][j]=0;
}
}

  /*** Do matrix multiply sharing iterations on outer loop ***/
  /*** Display who does which iterations for demonstration purposes ***/
printf("Thread %d starting matrix multiply...\n",tid);
#pragma omp for schedule (static, chunk)
for(i=0;i<a_r; i++)
{
//    printf("Thread=%d did row=%d\n",tid,i);
for(j=0;j<a_c; j++)
{
for(k=0;k<b_c; k++)
{
c[i][j]=c[i][j]+a[i][k]*b[k][j];
}
}
}


 } /*****end of parallel region*****/

printf("******************************************************\n");
printf ("Done.\n");
  double end = omp_get_wtime( );    //end the timer
  dif =  end - start; //store the difference

 printf("Parallelization took %f sec. time.\n", dif);
/*free memory*/
for(i=0;i<a_r; i++)
{
free(a[i]);
}
free(a);
for(i=0;i<a_c; i++)
{
free(b[i]);
}
free(b);
for(i=0;i<b_c; i++)
{
free(c[i]);
}
free(c);}
```